②

TECHNICAL REPORT RD-GC-93-16

APPLICATION OF IMAGE COMPRESSION TO DIGITAL MAP DATABASES

Marc W. Crooks
Guidance and Control Directorate
Research, Development, and Engineering Center

FEBRUARY 1993

U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35898-5000

Approved for public release; distribution is unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-RD-GC-93-16	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Guidance and Control Directorate RDEC 6b. OFFICE SYMBOL (If applicable) AMSMI-RD-GC-S	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Commander, U.S. Army Missile Command ATTN: AMSMI-RD-GC-S Redstone Arsenal, AL 35898-5254	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Application of Image Compression to Digital Map Databases

12. PERSONAL AUTHOR(S)
Marc W. Croo

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Feb 92 TO Sep 92	February 1993	38

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Image compression
Digitized paper maps

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Microprocessor Technology Utilization Program (RG-7) exists for the purpose of examining commercial microprocessor hardware and applying it to developing military systems prior to the release of any militarized components. As a part of this objective, RG-7 is looking at commercially available image compression chips/chipsets that might serve a valuable role in emerging military systems. Under the RG-7 program, we have examined still image compression chips/chipsets and their potential application in the area of digital map database compression.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Marc W. Crooks	22b. TELEPHONE (Include Area Code) (205) 842-6915 22c. OFFICE SYMBOL AMSMI-RD-GC-S

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Table of Contents

List of Figures

APPLICATION OF IMAGE COMPRESSION TO DIGITAL MAP DATABASES

I. INTRODUCTION

The Microprocessor Technology Utilization Program (RG-7) exists for the purpose of examining commercial microprocessor hardware and applying it to developing military systems prior to the release of any militarized components. As part of this objective, RG-7 is looking at commercially available image compression chips/chipsets that might serve a valuable role in emerging military systems. Under the RG-7 program, we have examined still image compression chips/chipsets and their potential application in the area of digital map database compression.

As imaging needs continue to grow, techniques are needed to help control the vast amounts of data produced. One example of this can be seen in the area of digitized paper maps. A single 1/50,000 scale ARC Digitized Raster Graphics (ADRG) map image requires approximately 373 Mbytes of memory to represent an area of approximately 44Km x 72Km. This memory requirement is clearly overwhelming to all but the largest storage media. To compound matters, when a significant area of coverage is required, even the largest storage media are ineffective. Not only are the memory requirements enormous, transmission of images this large would require extraordinarily long periods of time (4.5 days @ 9600 Baud). In order to reduce this massive storage and transmission requirement, this study examined the application of still image compression techniques to digitized paper map data.

II. BACKGROUND

Many military systems may soon begin to require the use of digitized paper maps for land navigation purposes. Some candidate systems may include Multiple Launch Rocket System (MLRS), Fiber-Optic Guided Missile (FOG-M), Unmanned Aerial Vehicle (UAV), and Unmanned Ground Vehicle (UGV). In this study, MLRS was the system targeted for implementing a compressed map database. When fielded, the MLRS Improved Fire Control System (IFCS) will contain a militarized 300-Mbyte hard disk for all system storage requirements. As previously stated, this will not even allow for a 44Km x 72Km area of coverage of digitized paper maps. Clearly, the solution is to reduce the storage size of the digitized paper maps while retaining high visual quality.

During the development of the MLRS IFCS, a Technical Risk Investigation System (TRIS) was designed to investigate problems associated with current technology limitations and explore possible solutions to these problems. As part of the MLRS TRIS, the use of compressed digitized paper maps was studied. This study looked at both the preprocessing phase (compression) of digitized paper maps as well as the decompression and display of these maps. As an end result of this study, it is hoped that lessons learned from the MLRS TRIS can be applied to any and all military systems requiring digitized paper map data.

III. IMAGE COMPRESSION TECHNIQUES

As part of this study, four types of image compression techniques were examined for their potential use in digitized paper map compression. These image compression techniques include Joint Photographic Experts Group (JPEG), Px64, Motion Picture Experts Group (MPEG), and Fractal Compression. See Table 1 for a brief summary of each compression technique. After careful examination of each data compression technique, it was decided that JPEG would be used to implement the compressed digitized paper map study. Two reasons were cited for this decision: (1) JPEG is currently in the balloting phase of becoming an international standard (ISO/IEC DIS 10918), and (2) vendors are currently marketing chips/chipsets that comply with the JPEG algorithm. At the time of this study, no chips/chipsets were found that implemented either the Px64 algorithm or MPEG algorithm. The fractal compression technique, although promising, is a proprietary algorithm and thus not an industry standard.

NAME	COMPRESSION ALGORITHM	INDUSTRY STANDARD	IMAGE TYPE	USEABLE RATIOS	REAL TIME COMP/DECOMP
JPEG	DCT	Balloting phase	Full color; Still images	25:1	Yes
Px64	DCT; Predictive interframe coding	Yes	Full color; Low motion level video images	100:1	Yes
MPEG	DCT; Interpolated interframe coding	Draft phase	Full color; Motion intensive video images	200:1	No-compression Yes-decompression
FRACTAL	Fractal Transform	No	Full color; Still images and motion intensive video images	76:1	No-compression Yes-decompression

Table 1. Image Compression Technique Comparison

IV. JPEG ALGORITHM

The JPEG algorithm is a Discrete Cosine Transform (DCT) based technique that is primarily used for continuous-tone still images. The algorithm itself relies heavily on reducing redundant visual information

DISCRETE COSINE TRANSFORM[1]

$$S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16}$$

by way of the DCT and quantization and then efficiently encoding the data. Because redundant data is removed during the JPEG compression process and therefore the compressed image no longer exactly matches the original image, this type of compression technique is said to be "lossy".

The sequence of events that occur during a JPEG compression can be seen in Figure 1. The first step involves performing a DCT on the image

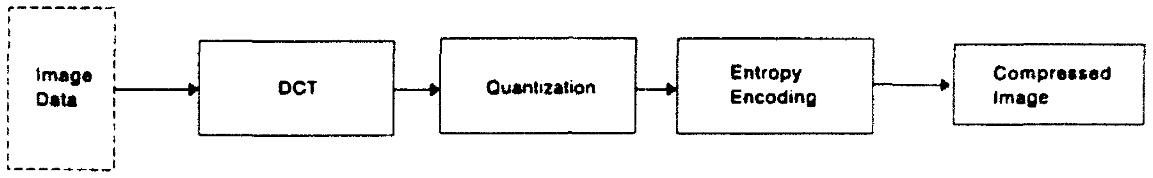


Figure 1. JPEG Baseline Compression

data. The DCT itself is used to transform the image data into its frequency coefficients. Once these frequency coefficients have been isolated, they are then quantized in order to reduce the range of possible frequency values. The level of quantization the frequency data undergoes is known as the "quantization factor". The quantization factor is a user definable value and, as will be discussed later, plays a significant role in the quality of a compressed image and the level of compression attained. The quantization phase of the compression process is the "lossy" portion of the JPEG algorithm. Entropy encoding then follows. This step compresses the data yet again but it does so in a "lossless" manner (data in exactly represents data out). Data that occurs frequently is represented using shorter codes while less frequently occurring data is represented with longer codes. The output from the encoding phase represents the compressed image.

In order to display a compressed image, it must first be decompressed. Because of the nature of the DCT and the JPEG standard, the sequence needed for decompression is simply the reverse of the compression process. The decompression sequence is illustrated in Figure 2. The compressed image is first decoded, an inverse quantization is then performed, and finally the Inverse Discrete Cosine Transform (IDCT) is used to convert the frequency coefficients back into pixel data.

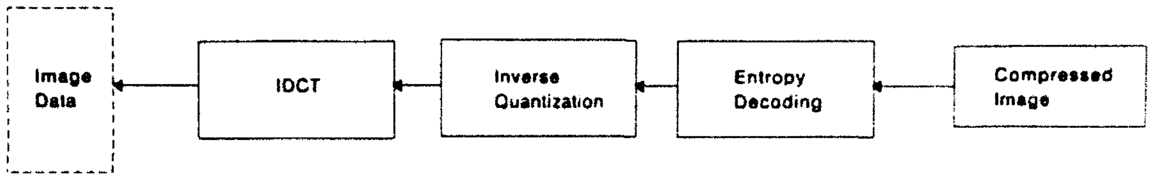


Figure 2. JPEG Baseline Decompression

The IDCT can be seen below.

INVERSE DISCRETE COSINE TRANSFORM[1]

$$s_{yx} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C_u C_v S_{vu} \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16}$$

V. EXTENSIONS TO THE JPEG STANDARD

The JPEG standard was written so as to perform compression on image data independent of the color space of the data. Vendors, however, have added specialized front ends to their chips/chipsets so as to allow the user to easily compress common image data formats. One such representation of image data is red, green, blue (RGB). Because many digital image databases are represented in this format, it was necessary for vendors to develop suitable front ends that would enable compression of RGB data. The most common front end solution to compressing RGB data is to convert it to its luminance and chrominance (YUV) components. This is done by performing the following linear transformation on the RGB data:

RGB-to-YUV TRANSFORMATION[2]

$$\begin{vmatrix} Y \\ U \\ V \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.3316 & 0.500 \\ 0.500 & -0.4186 & -0.0813 \end{vmatrix} \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

Upon decompression, the inverse of the above matrix is used to go from YUV color space to RGB. Figure 3 shows the JPEG baseline compression/decompression algorithm plus extensions provided by vendors for handling RGB color space issues. Color spaces other than RGB that vendors make allowances for include cyan, magenta, yellow, black (CMYK) and gray scale.

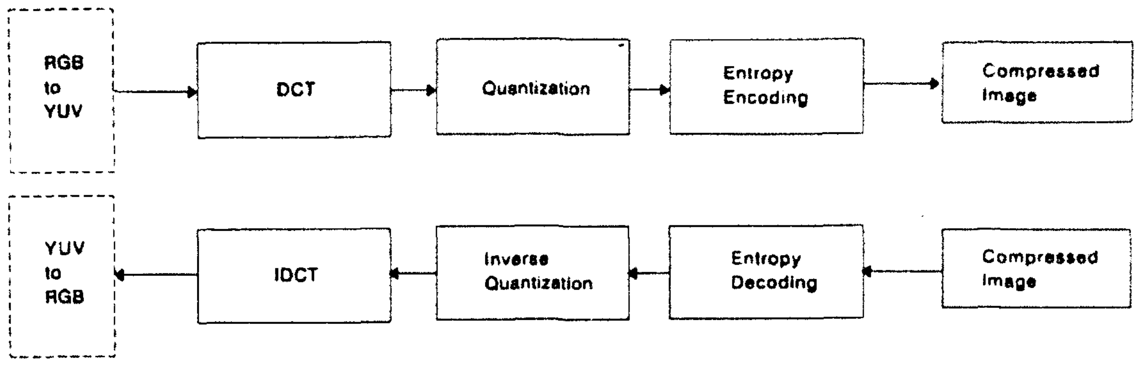


Figure 3. JPEG Baseline Algorithm Plus Color Space Extension

VI. DATABASE PREPROCESSING

In order to adequately examine digitized paper map database compression/decompression, a suitable database had to be selected that would comply with the requirements imposed by the MLRS IFCS. The database selected was 1/50,000 scale ADRG. This database is distributed by the Defense Mapping Agency (DMA) and is available on compact disk (CD-ROM). The area of coverage received was from 49^o 00'N to 49^o 24'N longitude and 11^o 00'E to 12^o 00'E latitude (part of what was formerly West Germany). This digitized paper map data is in 24-bit RGB format and requires approximately 373 Mbytes of space on the CD-ROM.

The format of the image data on the CD-ROM required that the data be reformatted prior to compressing the image. As shown in Figure 4 the raw image data is broken down into 128x128 pixel tiles. Each of these tiles

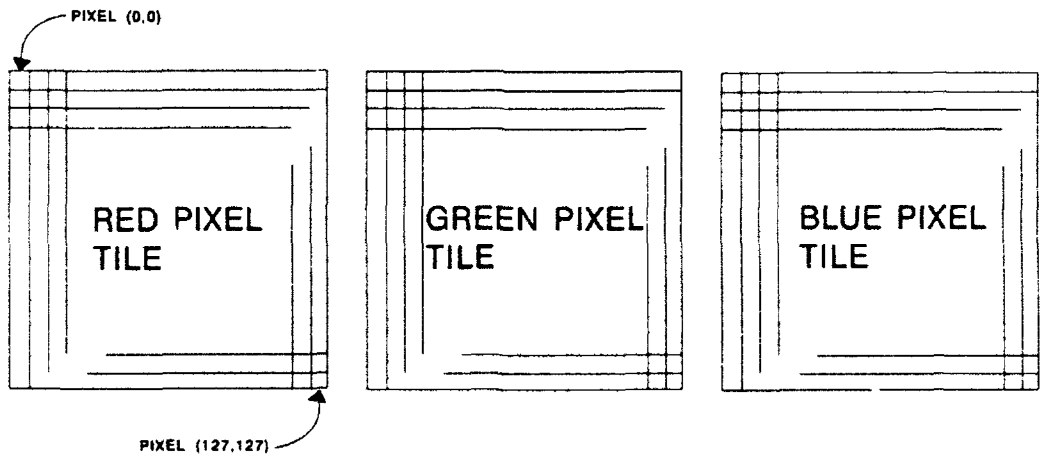


Figure 4. ADRG Pixel Tile Arrangement

represents either red color byte information, green color byte information, or blue color byte information. For proper compression, the image data must be in a sequential RGB format. This means that for any image tile, red $\text{pixel}_{x,y}$ must be placed next to green $\text{pixel}_{x,y}$ which should be adjacent to blue $\text{pixel}_{x,y}$. All pixels in each image tile are arranged in this RGB manner prior to any compression. RGB-ordered image tiles are then saved to a hard drive to await compression. Code that was generated to perform this reformatting of the ADRG image data can be found in Appendix A.

Once the complete ADRG image has been reformatted, the compression portion of the database preprocessing sequence can begin. This sequence prompts the user for the upper left longitudinal and latitudinal coordinates of the image, the compressed file prefix, and the quantization factor that will be used in compression of all image tiles. Once determined, all image tiles will be named using the user inputs and compressed using the particular quantization value. The quantization factor is used to determine the level of redundancy eliminated and **does not** directly represent the compression ratio that will be achieved. Because some ADRG image tiles will contain more redundant visual information than others, the compression levels between tiles will differ. After each image tile has been compressed, it is then stored on the MLRS TRIS hard drive until needed for display. Figure 5 shows the entire preprocessing sequence as performed on a 486 computer equipped with a CD-ROM.

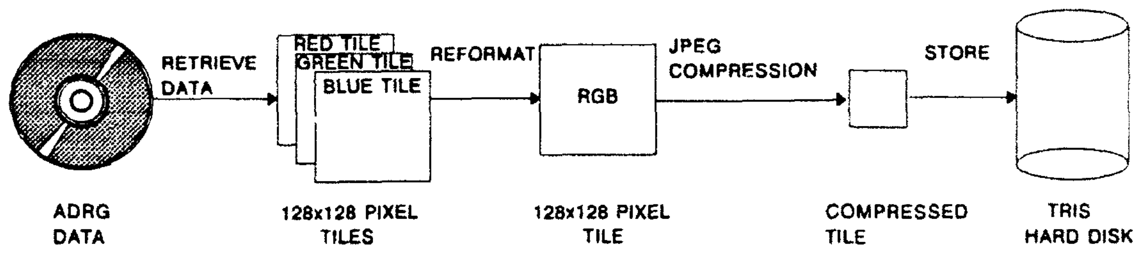


Figure 5. ADRG Database Preprocessing

VII. DATABASE DECOMPRESSION

An important consideration in database decompression was to make sure that images were decompressed quickly so as to not create huge latency periods during data updates. From Table 1 it can be seen that JPEG compliant chips/chipsets will perform image compression/decompression in "real-time". This means that images can be compressed/decompressed at video rates of 30 frames per second. This claim of "real-time" can be somewhat misleading because it is dependent upon the resolution and quantization level of the image. An image with less resolution can be compressed at a lower quantization level (lower compression ratio; better visual quality) in "real-time" while an image with a higher resolution must be compressed at higher quantization levels (higher compression ratio; poorer visual quality) in order to be "real-time". A second point must also be made about compression/decompression rates. Even though compression/decompression may occur at rates of 30 frames per second, one must still transfer data to and from the compression/decompression card. This means that a bottleneck exits with respect to transferring data to and from a hard drive, memory, serial input/output (I/O), CD-ROM, etc. In order for the compression chip/chipset to achieve maximum compression/decompression rates, high data transfer rates must be supported and this is sometimes not achievable in a personal computer (PC) environment. The "real-time" compression/decompression claim of JPEG compliant chip/chipset vendors must be viewed with an understanding of the end goal and the possible limitations of the target system.

With this in mind, we remember that in our application we are examining the use of JPEG to compress/decompress digitized paper map data for the purposes of land navigation. For our study, this task does not need to be performed in "real-time". A vehicle such as MLRS is not overly quick and will perhaps sit at a single location for long periods of time. This will not require high update rates from the compressed image data. Perhaps the only time that a high volume of data will be required for decompression and display will be at startup. At this time, an initial map will be retrieved from memory, decompressed, and displayed. After that, only small updates will be required to support vehicle movement, etc. The "real-time" decompression and update of the map is not a critical issue. The decompression and display of the map data should be reasonably quick but "real-time" is not a requirement.

The decompression side of digitized map database display involves three main events. These three events include map tile retrieval, decompression, and display. Each of these events must occur in a manner that will minimize the overall time required for map updates. For demonstration purposes, code was generated on a 486 PC that retrieved, decompressed, and displayed compressed ADRG data (see Appendix A).

Retrieval of map involves first selecting a tile or multiple tiles for display. This might involve prompting a user for input or using navigation system data to select map data for display. The map data could then be retrieved from some sort of storage device (hard drive, CD-ROM, etc.). For demonstration purposes, manual input of a map tile was required. Once received, adjacent tiles (up to 2 rows/2 columns away) were determined. This produced a 5x5 matrix of map tile data with a resolution of 640x640 pixels (height = 5*128, width = 5*128).

Once the tiles needed for display were determined, decompression of the tiles could begin. Tiles were retrieved sequentially starting with $\text{tile}_{0,0}$ of the 5x5 matrix and ending with $\text{tile}_{4,4}$. Retrieval and decompression of each tile required less than .5 seconds for completion. This decompression time resulted from tests using LEAD Technology's LEADVIEW 255 compression card based on the C-Cube CL550 compression processor and LEAD Technology's LEADTOOLS V2.3 software run-time. Again, most of this time can be attributed to hard drive accesses and data transfers across the PC bus rather than the actual decompression operation itself.

As stated earlier, all tiles were compressed from an RGB (24-bit color) format. Once decompressed the tiles needed to be color optimized for display. This is due to the fact that Video Graphics Adapter (VGA) displays only support a maximum of 256 colors. RGB color provides up to a maximum of 16.7 million colors. Color optimization was used to select the best 256 colors needed to reproduce each image tile. Examining this a little further reveals that for each image tile there exists 256 colors that will best represent each tile; however, each tile may not optimize to the same 256 colors. This means that in order to achieve the best 256 colors for the entire displayed image (5 tiles x 5 tiles), the color palettes for each tile must be optimized as a whole in order to get the best 256 colors that represent the entire image. After decompressing each tile, a color optimization was performed on the tile. The optimized tile palette was then stored in a file containing all other optimized tile palettes. After completing the optimization on all 25 tiles, the optimized palette file was then itself optimized to 256 colors. This color optimization produced the "image palette".

Display of the map tiles involved indexing all 8-bit pixel color values from each tile into the "image palette". All image tiles were then displayed on a VGA monitor. For any tile additions required for display updates, a new "image palette" would have to be created in order to obtain accurate color representations for all the map tiles.

VIII. RESULTS

The results of this study reveal several important points regarding compressed database preparation/display and image quality versus compression ratios achieved.

As evidence from this study, it is possible to produce compressed map databases with a minimal amount of hardware. In fact, the only specialized equipment needed was the actual compression/decompression board. By implementing database preparation on such a simple system, two important objectives are met. The first objective is to produce a database preparation system that is low cost. A complete ADRG database compression system should cost no more than $10,000 including the compression/decompression card. The second objective is to maintain compatibility with available systems. Because the entire ADRG database preparation system runs on an MS-DOS based PC, all MS-DOS systems (with proper hardware) should be able to perform ADRG database preparation either in the field or in the office.

During the course of this study various quantization factors were examined in order to determine their effect on compression ratios. By doing this, we were able to ascertain the compression levels that could typically be expected on ADRG data. Figure 6 shows various quantization factors and the actual compression ratio achieved using that factor. These compression

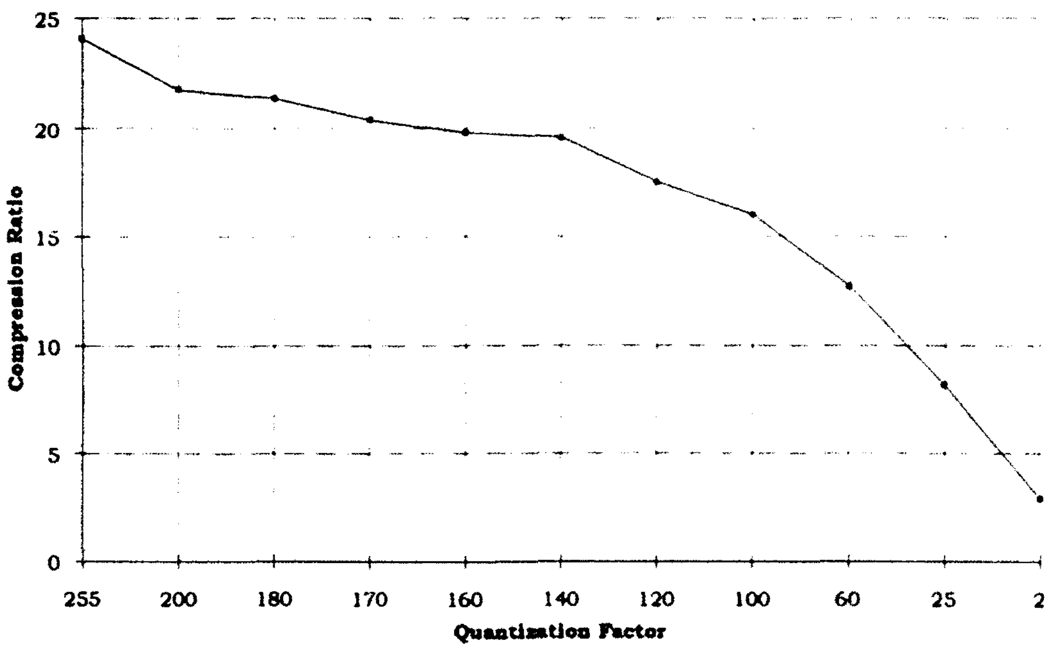


Figure 6. ADRG Compression Ratios vs. Quantization Factors

ratios by no means represent the level of compression that will be achieved for all ADRG database compression. It is simply a gauge of the levels that one might expect after compressing an ADRG database. Results show that the highest quantization factor (Q = 255), produces compression ratios of approximately 25:1. More realistic compression ratios (from a visual standpoint) of approximately 20:1 are obtainable using quantization factors ranging from Q = 140 through Q = 170. Images compressed using quantization factors below 100 yield lower compression ratios but better quality images.

When discussing image compression, another very important consideration that must be addressed is the issue of compression ratio versus image quality. This is a very difficult and highly subjective issue to discuss. An image that may appear pleasing to one individual may appear unacceptable to another. Although JPEG is designed to reduce visually redundant information so as to be less perceivable to the human eye, when compression ratios are increased to a high enough level, visual distortions begin to appear. The question then becomes how much distortion is too much? No quantitative measurement was located that can state how much visual degradation is too much for an individual. It is purely a subjective decision. Figure 7 shows a map section that has not been compressed. Compare this to Figure 8 which shows the same map section after being compressed using a quantization factor of 2 (compression ratio = 2.9:1). Next compare these figures with Figure 9 which has been compressed using a quantization factor of 160 (compression ratio = 19.8:1). Finally, Figure 10 shows the same image after it has been compressed using the maximum quantization factor of 255 (compression ratio

= 24.1:1). Clearly, as the quantization factor increases, the visual degradation of the image increases.

IX. CONCLUSIONS

This study has shown that image compression is certainly a very useful tool in managing the size of huge image databases. Using JPEG compression techniques can result in images being compressed at ratios of approximately 20:1 while still maintaining good visual quality. This compression/decompression can also be done in a timely manner due to chips/chipsets currently available. This is important to systems that have stringent timing requirements.

This study has also shown that database preparation systems can be both inexpensive and easily implemented using common MS-DOS based PCs and an image compression/decompression card.

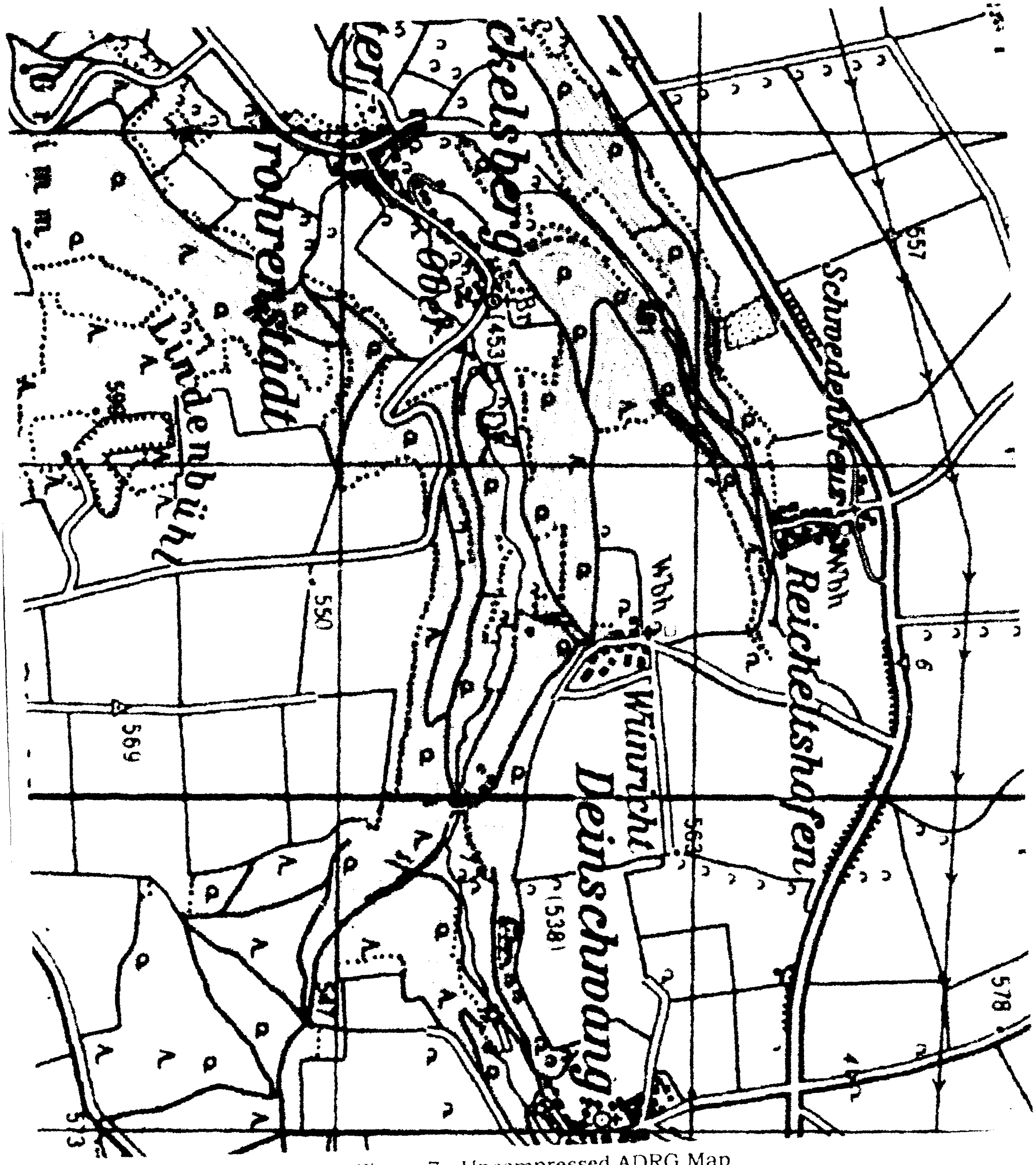


Figure 7 Uncompressed ADRG Map

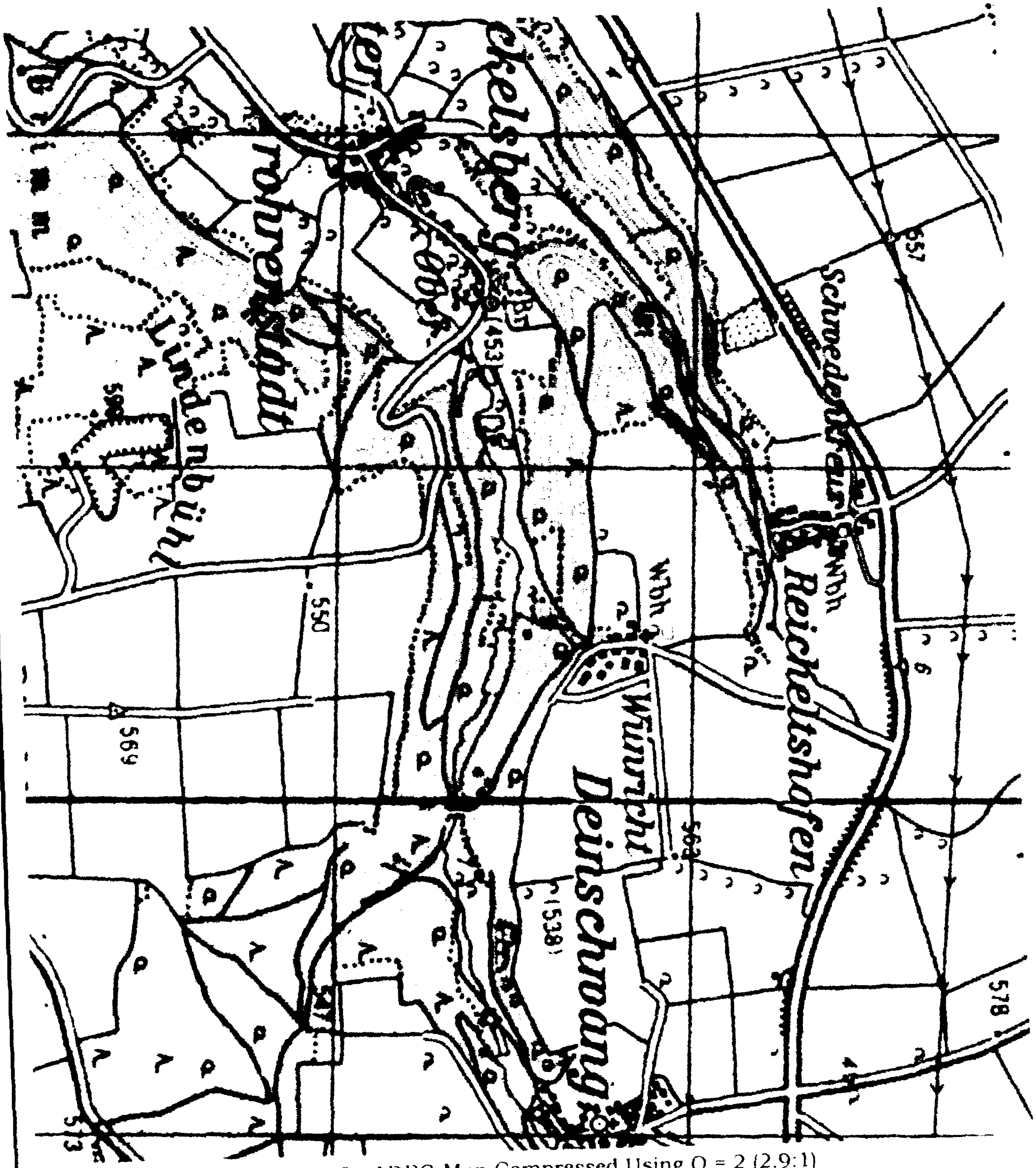


Figure 8. ADRG Map Compressed Using Q = 2 (2.9:1)

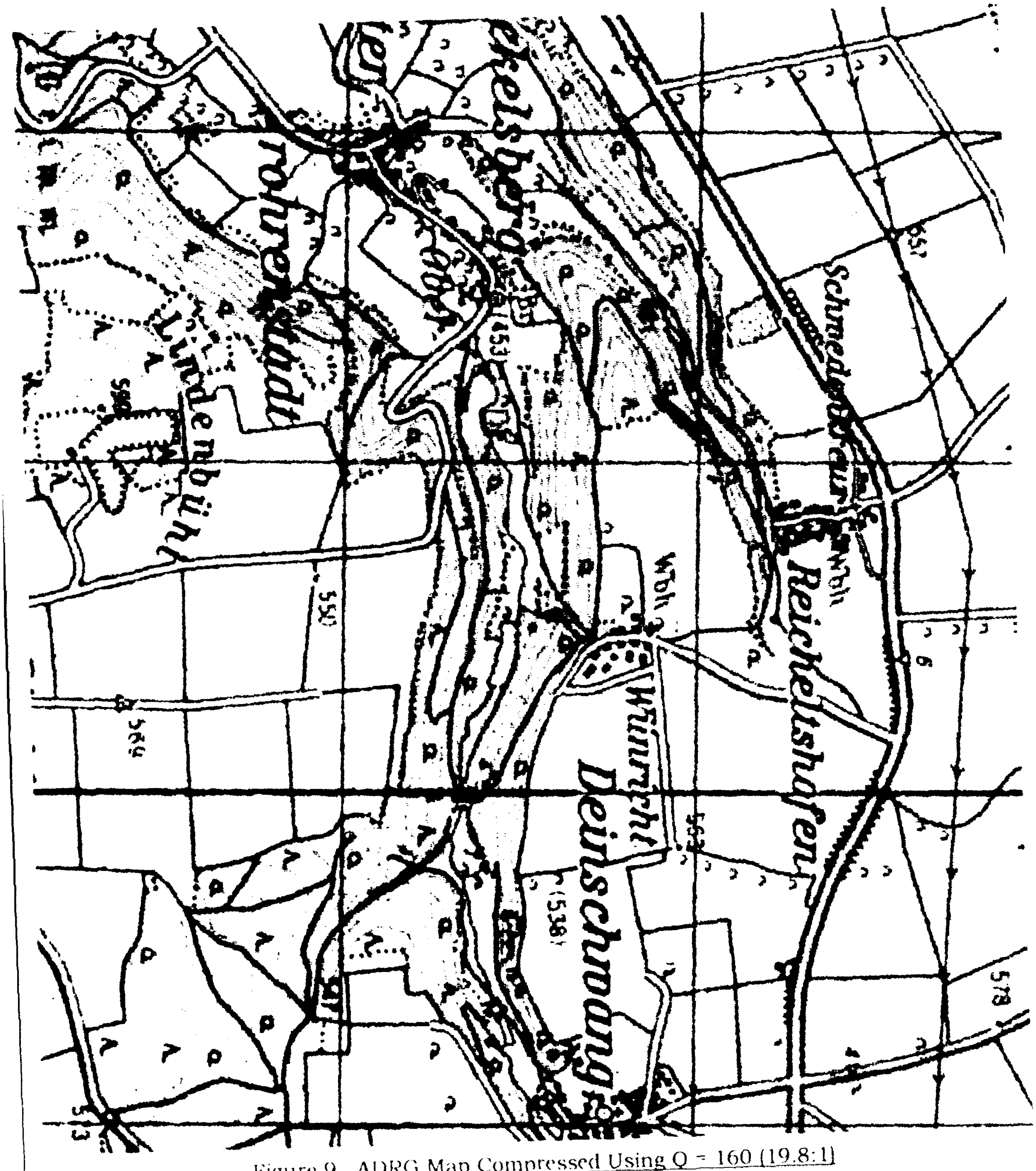


Figure 9. ADRG Map Compressed Using Q = 160 (19.8:1)

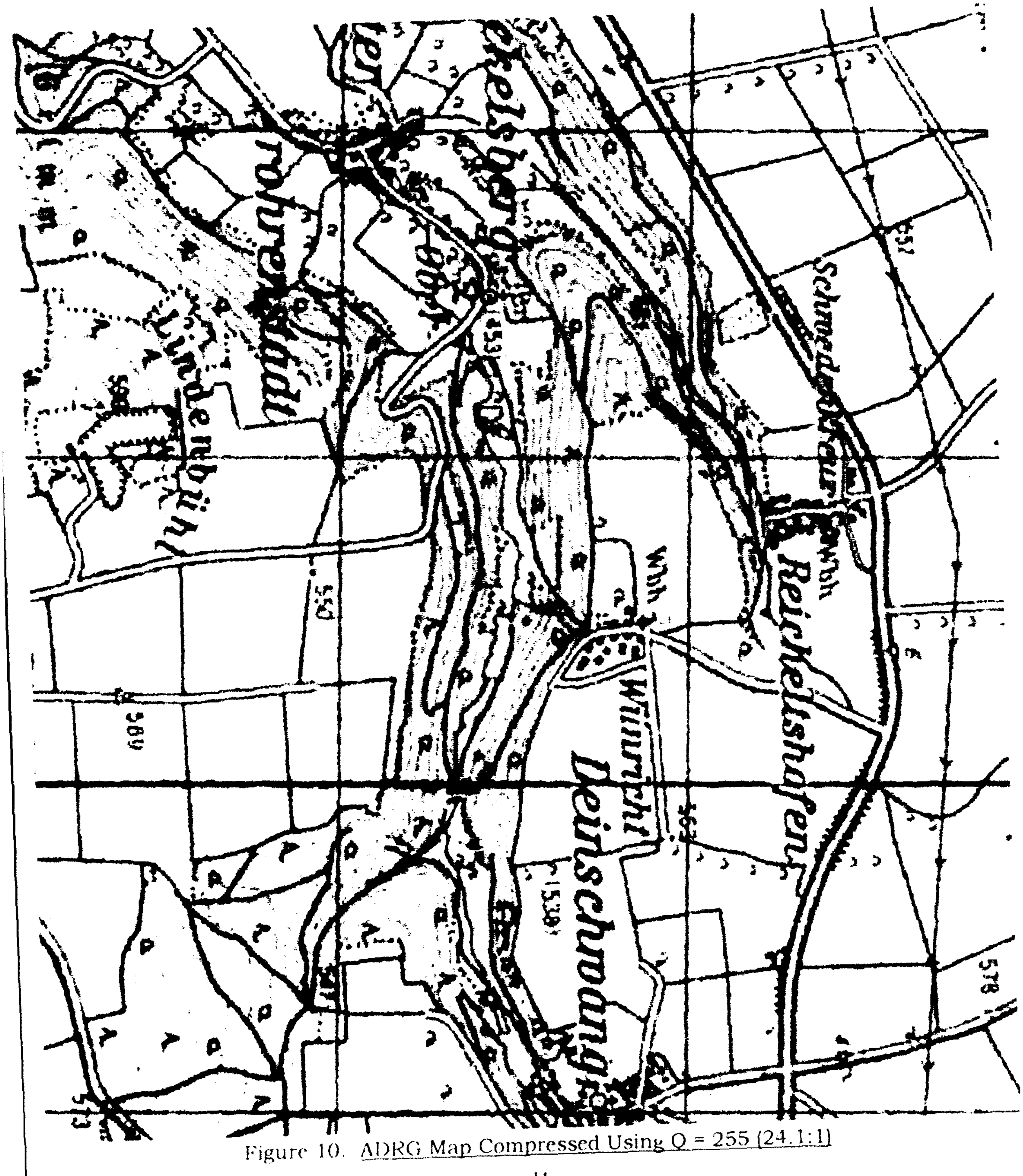


Figure 10. ADRG Map Compressed Using Q = 255 (24.1:1)

APPENDIX A

CODE TO REFORMAT ADRG DATA

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <math.h>
#include <errno.h>

#define true 1
#define false 0

char red_buff[16384];
char *red_buff_ptr = red_buff;
char green_buff[16384];
char *green_buff_ptr = green_buff;
char blue_buff[16384];
char *blue_buff_ptr = blue_buff;
char adrg_img_file[13];
char adrg_gen_file[13];
char path_name[28];
static char long_ul[11];
static char lat_ul[10];

void main()
{
static unsigned long Asz, Bs;
static unsigned long NUL, NUS, NLL, NLS;
char *buffer_ptr;
static unsigned int image_height, image_width;
static char cell_name[13];
static int cnt;
static int read_handle;
static int read_handle_img;
static int write_handle;
unsigned int index = 16384;
char buffer;
static char row_num[3];
static char col_num[3];
char user_response[1];
static int i = 0;
static int j = 0;
int invalid_response = true;

/* This portion of the code prompts the user for input regarding the name  */
/* of the ADRG General Information File and the path of the file.  The     */
/* code also prompts the user for verification of input.                   */
     while(invalid_response == true){
        clrscr();
        printf("Name of ADRG General Information File (filename.ext): ");
        gets(adrg_gen_file);
        printf("Is (%s) Correct? (Y or N): ", adrg_gen_file);
        gets(user_response);
        if((user_response[0] == 'Y') || (user_response[0] == 'y')){
           printf("Input Path Name (d:\\pathname\\): ");
           gets(path_name);
           printf("Is (%s) Correct? (Y or N): ", path_name);
           gets(user_response);
           if((user_response[0] == 'Y') || (user_response[0] == 'y'))
              invalid_response = false;
        }
      }

/* Combine the path with the General Information File name and open the    */
/* file.  Discard the first 1777 bytes of information in the file.  These   */
/* bytes may be useful to some applications.  Consult the ADRG Product      */
/* Specification for more information.                                      */
      i = 0;
      while(path_name[i] != '\0')
        i++;
      for(j = 0; adrg_gen_file[j] != '\0'; j++)
        path_name[i++] = adrg_gen_file[j];
      path_name[i] = '\0';

      buffer_ptr = &buffer;
      read_handle = open(path_name, O_RDONLY | O_BINARY);
      read(read_handle, buffer_ptr, 1777);

/* Read the 8 bytes that represent the east-west pixel spacing and convert */
/* that value to a long (Asz).                                              */
      buffer_ptr = &Asz;
      read(read_handle, buffer_ptr, 8);
      Asz = atol(buffer_ptr);

/* Read the 8 bytes that represent the north-south pixel spacing and       */
/* convert that value to a long (Bs).                                       */
      buffer_ptr = &Bs;
      read(read_handle, buffer_ptr, 8);
      Bs = atol(buffer_ptr);

/* Read the 11 bytes that represent the longitudinal coordinates of the    */
/* upper left corner of the area of coverage into the long_ul array.  The  */
/* format is +/- DDDMMSS.SS.                                                 */
      buffer_ptr = &long_ul;
      read(read_handle, buffer_ptr, 11);

/* Read the 10 bytes that represent the latitudinal coordinates of the     */
/* upper left corner of the area of coverage into the lat_ul array.  The   */
/* format is +/- DDMMSS.SS.                                                  */
      buffer_ptr = &lat_ul;
      read(read_handle, buffer_ptr, 10);
```

```
/* Read and discard the next 65 bytes.  These bytes may useful to some     */
/* applications so code modification may be necessary.                    */
     buffer_ptr = &buffer;
     read(read_handle, buffer_ptr,65);

/* Read the 6 bytes that represent the row number of the upper right       */
/* corner of image data (NUL).  The value is then converted to a long.     */
     buffer_ptr = &NUL;
     read(read_handle, buffer_ptr, 6);
     NUL = atol(buffer_ptr);

/* Read the 6 bytes that represent the column number of the upper right    */
/* corner of image data (NUS).  The value is then converted to a long.     */
     buffer_ptr = &NUS;
     read(read_handle, buffer_ptr, 6);
     NUS = atol(buffer_ptr);

/* Read the 6 bytes that represent the row number of the lower left        */
/* corner of image data (NLL).  The value is then converted to a long.     */
     buffer_ptr = &NLL;
     read(read_handle, buffer_ptr, 6);
     NLL = atol(buffer_ptr);

/* Read the 6 bytes that represent the column number of the lower left     */
/* corner of image data (NLS).  The value is then converted to a long.     */
     buffer_ptr = &NLS;
     read(read_handle, buffer_ptr, 6);
     NLS = atol(buffer_ptr);

/* Read the 3 bytes that represent the height of the map image (in 128x128 */
/* pixel tiles).  The value is then converted to an integer.               */
     buffer_ptr = &image_height;
     read(read_handle, buffer_ptr, 3);
     image_height = atoi(buffer_ptr);

/* Read the 3 bytes that represent the width of the map image (in 128x128  */
/* pixel tiles).  The value is then converted to an integer.               */
     buffer_ptr = &image_width;
     read(read_handle, buffer_ptr, 3);
     image_width = atoi(buffer_ptr);

/* Read and discard the next 17 bytes.  These bytes may useful to some     */
/* applications so code modification may be necessary.                    */
     buffer_ptr = &buffer;
     read(read_handle, buffer_ptr, 17);

/* Read the 12 bytes that represent the name of the ADRG Image File.  The  */
/* bytes are stored in the adrg_img_file array.                            */
     buffer_ptr = &adrg_img_file;
     read(read_handle, buffer_ptr, 12);

/* Read and discard the next 77 bytes.  These bytes may useful to some     */
/* applications so code modification may be necessary.                    */
     buffer_ptr = &buffer;
```

```
    read(read_handle, buffer_ptr, 77);

  /* Open the ADRG image file and strip out unused header bytes.  Color     */
  /* tile information is then stored for reformatting process.              */
    i = 12;
    for(j = 0; adrg_img_file[j] != '\0'; j++)
     path_name[i++] = adrg_img_file[j];

    read_handle_img = open(path_name, O_RDONLY | O_BINARY);
    read(read_handle_img, buffer_ptr, 2048);
    j = 0;
    i = 0;
    printf("\n");

/****************************************************************************/
/* This is the loop that names and reformats every image tile in every     */
/* row.  The image_height and image_width variables are used to keep track */
/* of this loop.  Tiles are named with the following convention:           */
/*                                                                          */
/*              TILE NOMENCLATURE - DDD dd RRR.CCC                          */
/*                                                                          */
/*                  DDD - Longitude (degrees) of the upper left map corner  */
/*                  dd  - Latitude (degrees) of the upper left map corner   */
/*                  RRR - Map tile row number                               */
/*                  CCC - Map tile column number                            */
/****************************************************************************/
    for(i = 0; (i < image_height); i++){
     for(j = 0; j < image_width; j++){

         for(cnt = 0; cnt < 3; cnt++)
            cell_name[cnt] = long_ul[cnt + 1];
         for(cnt = 3; cnt < 5; cnt++)
            cell_name[cnt] = lat_ul[cnt - 2];
         itoa(i, row_num, 10);
         if(i <= 9){
            cell_name[5] = '0';
            cell_name[6] = '0';
            cell_name[7] = row_num[0];
         }
         else if(i <= 99){
            cell_name[5] = '0';
            cell_name[6] = row_num[0];
            cell_name[7] = row_num[1];
         }
         else{
            for(cnt = 5; cnt < 8; cnt++)
              cell_name[cnt] = row_num[cnt - 5];
         }
         cell_name[8] = '.';
         itoa(j, col_num, 10);
         if(j <= 9){
            cell_name[9] = '0';
            cell_name[10] = '0';
            cell_name[11] = col_num[0];
            }
            else if(j <= 99){
               cell_name[9] = '0';
               cell_name[10] = col_num[0];
               cell_name[11] = col_num[1];
            }
            else{
               for(cnt = 9; cnt < 12; cnt++)
                 cell_name[cnt] = col_num[cnt - 9];
            }
            cell_name[12] = '\0';

            index = 16384;
            printf("Now formatting %s\n ", cell_name);

   /* This section reads the data from the image tile into 3 buffers.  These  */
   /* are the red, green, and blue color byte buffers for each tile.          */
            write_handle = open(cell_name, O_CREAT | O_BINARY, S_IREAD
                 | S_IWRITE);
            read(read_handle_img, red_buff_ptr, index);
            read(read_handle_img, green_buff_ptr, index);
            read(read_handle_img, blue_buff_ptr, index);

    /* Create files for the 128x128 pixel blocks.  Pixels are then written to */
    /* a file in the format R-byte, G-byte, B-byte until all 128x128x24 pixel */
    /* values have been written.                                               */
            for(index = 0; index < 16384; index++){
               write(write_handle, red_buff_ptr++, 1);
               write(write_handle, green_buff_ptr++, 1);
               write(write_handle, blue_buff_ptr++, 1);
            }
            red_buff_ptr = &red_buff;
            green_buff_ptr = &green_buff;
            blue_buff_ptr = &blue_buff;
            close(write_handle);
         }
       }
       close(read_handle);
       close(read_handle_img);
       return(0);
   }
```

CODE TO COMPRESS ADRG DATA

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <fcntl.h>
#include <sys\stat.h>
#i .lude <sys\types.h>
#include <io.h>
#include <math.h>
#include <errno.h>
#include <toolapp.h>
#include <l_error.h>
#include <l_bitmap.h>

#define true 1
#define false 0

/* Increase the size of the stack.                                          */
extern unsigned _stklen = 14000U;

void main()
{
     BITMAPHANDLE ADRG_Bitmap;
     int Q_factor;
     unsigned int image_height = 71;
     unsigned int image_width = 107;
     static char long_ul[11];
     static char lat_ul[10];
     char row_num[3];
     char col_num[3];
     static char cell_name[15];
     static int i, j, cnt;
     int read_handle;
     unsigned char *pBuf;
     int ret_val;
     char user_response[5];
     int invalid_response = true;
     char file_prefix[5];

/* Prompt the user for the longitude and latitude of the upper left map     */
/* corner, the file prefix (for naming purposes), and the Q factor to be    */
/* used for compression.                                                    */
     while(invalid_response == true){
       clrscr();
       printf("Input Longitude of the Upper Left Corner (DDD): ");
       gets(long_ul);
       printf("Input Latitude of the upper left corner (dd): ");
       gets(lat_ul);
       printf("Input File Prefix (A-Z): ");
       gets(file_prefix);
       printf("Input Q Factor for File Compression (1-255): ");
       gets(user_response);
```

```
         Q_factor = atoi(user_response);
         printf("Is %d Correct? (Y or N): ", Q_factor);
         gets(user_response);
         if((user_response[0] == 'Y') || (user_response[0] == 'y'))
            invalid_response = false;
      }

/* Compress files within the boundaries given by the map image height and  */
/* width.                                                                   */
      for(i = 0; i < image_height; i++){
         for(j = 0; j < image_width; j++){

/* Create the name of the file to be compressed using the user input for   */
/* upper left longitude (long_ul) and upper left latitude (lat_ul).
            for(cnt = 0; cnt < 3; cnt++)
               cell_name[cnt] = long_ul[cnt];
            for(cnt = 3; cnt < 5; cnt++)
               cell_name[cnt] = lat_ul[cnt - 3];

/* Convert image row index (i) to a string and store it in the row_num     */
/* array for use in generating a file name.  As the row number is          */
/* incremented, the cell_name will be incremented accordingly.             */
            itoa(i, row_num, 10);
            if(i <= 9){
               cell_name[5] = '0';
               cell_name[6] = '0';
               cell_name[7] = row_num[0];
            }
            else if(i <= 99){
               cell_name[5] = '0';
               cell_name[6] = row_num[0];
               cell_name[7] = row_num[1];
            }
            else{
               for(cnt = 5; cnt < 8; cnt++)
                 cell_name[cnt] = row_num[cnt - 5];
            }
            cell_name[8] = '.';

/* Convert image column index (j) to a string and store it in the col_num  */
/* array for use in generating a file name.  As the column number is       */
/* incremented, the cell_name will be incremented accordingly.             */
            itoa(j, col_num, 10);
            if(j <= 9){
               cell_name[9] = '0';
               cell_name[10] = '0';
               cell_name[11] = col_num[0];
            }
            else if(j <= 99){
               cell_name[9] = '0';
               cell_name[10] = col_num[0];
               cell_name[11] = col_num[1];
            }
            else{
```

```
            for(cnt = 9; cnt < 12; cnt++)
              cell_name[cnt] = col_num[cnt - 9];
        }

/* Close cell_name array with the end-of-string character '\0'.           */
        cell_name[12] = '\0';

/* Initialize and allocate space (in expanded memory) for a 128x128 pixel  */
/* (24 bit-per-pixel) image bitmap.  */
        L_InitBitmap(&ADRG_Bitmap, 128, 128, 24);
        L_AllocateBitmap(&ADRG_Bitmap, TYPE_NOCONV);

/* Open the map file and load each pixel row into the bitmap using the     */
/* L_PutBitmapRow call.  If no file exists (read_handle < 0) then exit the */
/* loop.                                                                   */
        if((read_handle = open(cell_name, O_RDONLY | O_BINARY)) > 0){
            for(cnt = 0; cnt < 128; cnt++){
              read(read_handle, pBuf, 384);
              L_PutBitmapRow(&ADRG_Bitmap, pBuf, cnt, 384);
            }

/* Close the file.                                                         */
            close(read_handle);

/* Insert the file prefix into the first position of the cell_name array.  */
/* Compress the image and report whether the compression was successful.   */
            cell_name[0] = file_prefix[0];
            ret_val = L_CompressBitmap(&ADRG_Bitmap, cell_name, JFIF,
                  Q_factor, NoVGAPalette);
            if(ret_val == SUCCESS)
              printf("Compression on file %s was a success\n", cell_name);
            else
              printf("Error #%d occurred\n", ret_val);
        }
        else
            printf("Unable to Open File %s\n", cell_name);

/* Free the bitmap using L_FreeBitmap and begin compression on the next    */
/* file.                                                                   */
        L_FreeBitmap(&ADRG_Bitmap);
      }
    }
}
```

CODE TO DECOMPRESS AND DISPLAY ADRG DATA

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <math.h>
#include <errno.h>
#include <graphics.h>
#include <alloc.h>
#include <toolapp.h>
#include <l_error.h>
#include <l_bitmap.h>

#define true 1
#define false 0

/* Increase the size of the stack.                                         */
extern unsigned _stklen = 14000U;

void main()
{
     BITMAPHANDLE ADRG_Bitmap, New256_Color, New_Palette, Old_Palette ;
     char row_num[5];
     char display_tile[15];
     char dcom_tile[15];
     unsigned int cnt;
     unsigned int index;
     signed j, i;
     int read_handle, write_handle;
     int x_coord;
     int y_coord;
     char pixel_color;
     char center_tile[15];
     int invalid_response = true;
     char user_response[10];
     signed int tile_row_cnt;
     signed int tile_col_cnt;
     int x_center = 256;
     int y_center = 176;
     signed int image_width = 107;
     signed int image_height = 71;
     unsigned char pBuf_array[128];
     unsigned char *pBuf = pBuf_array;
     unsigned char new_pBuf_array[256];
     unsigned char *new_pBuf = new_pBuf_array;
     unsigned char pRow_array[768];
     unsigned char *pRow = pRow_array;

     char col_num[5];
     int ret_val;
     unsigned int display_cnt;
     unsigned int p_row_cnt = 0;

     char far *fptr;
     unsigned long mem_size;
     unsigned char color_index;
     int mode, page;

     char first_char[1];

/* Store current video mode.                                            */
     L_GetVideoMode(&mode, &page);

/* Prompt and verify the user for the center tile image to be displayed.   */
     while(invalid_response == true){
       clrscr();
       printf("Input Center Tile (LonLtRow.Col): ");
       gets(center_tile);
       printf("Is %s Correct? (Y or N): ", center_tile);
       gets(user_response);
       if((user_response[0] == 'Y') || (user_response[0] == 'y'))
          invalid_response = false;
     }

/* Create a file for storage of color palettes.                          */
     write_handle = open("Palette.mwc", O_CREAT | O_BINARY | O_RDWR, S_IREAD |
          S_IWRITE);

/* Extract row and column information from the center tile name.         */
     for(index = 5; index < 8; index++)
       row_num[index - 5] = center_tile[index];
     for(index = 9; index < 12; index++)
       col_num[index - 9] = center_tile[index];
     tile_row_cnt = atoi(row_num);
     tile_col_cnt = atoi(col_num);

/* Remove the file prefix and replace with the original 0 found in the     */
/* longitudinal value of the upper left map corner.                       */
     first_char[0] = center_tile[0];
     center_tile[0] = 0;

/* Load the display_tile array with the longitudinal and latitudinal      */
/* values that make-up the center_tile name.                              */
     for(index = 0; index < 5; index++)
       display_tile[index] = center_tile[index];

/* This loop is used to display a 5x5 matrix of tiles that will be        */
/* centered around the tile input by the user.  The first tile processed  */
/* is in the upper left of the image.  The tiles along that row are then  */
/* processed sequentially and the next row is then processed starting from */
/* the left side and working right.                                       */
     for(i = tile_row_cnt - 2; i <= tile_row_cnt + 2; i++){
       itoa(i, row_num, 10);
       if((i < 0) || (i >= image_height)){
         for(index = 5; index < 8; index++)
            display_tile[index] = '?';
      }
      else if(i <= 9){
         display_tile[5] = '0';
         display_tile[6] = '0';
         display_tile[7] = row_num[0];
      }
      else if(i <= 99){
         display_tile[5] = '0';
         display_tile[6] = row_num[0];
         display_tile[7] = row_num[1];
      }
      else{
         for(cnt = 5; cnt < 8; cnt++)
            display_tile[cnt] = row_num[cnt - 5];
      }

      for(j = tile_col_cnt - 2; j <= tile_col_cnt + 2; j++){
         itoa(j, col_num, 10);
         display_tile[8] = '.';

/* If the tile is out of bounds (ie. the column number is less than 0 and  */
/* greater than the image_width) a '?' is loaded into the file name so as  */
/* to produce a nonexistent tile name that will not be displayed.          */
         if((j < 0) || (j >= image_width)){
            for(index = 9; index < 12; index++)
              display_tile[index] = '?';
         }
         else if(j <= 9){
            display_tile[9] = '0';
            display_tile[10] = '0';
            display_tile[11] = col_num[0];
         }
         else if(j <= 99){
            display_tile[9] = '0';
            display_tile[10] = col_num[0];
            display_tile[11] = col_num[1];
         }
         else{
            for(cnt = 9; cnt < 12; cnt++)
              display_tile[cnt] = col_num[cnt - 9];
         }

/* Close cell_name array with the end-of-string character '\0'.            */
         display_tile[12] = '\0';

/* Load the dcom_tile array with the display_tile array.                   */
         for(cnt = 0; cnt < 13; cnt++)
            dcom_tile[cnt] = display_tile[cnt];

/* Add the file prefix to the display_tile array in position [0]           */
         display_tile[0] = first_char[0];
```

```
/* Initialize a bitmap of 128x128 pixels and 24-bit color                   */
        L_InitBitmap(&ADRG_Bitmap, 128, 128, 24);

/* Decompress the display_tile and store it in the bitmap                   */
        printf( "Decompression on tile %s", display_tile ) ;
        ret_val = L_DecompressBitmap(display_tile, &ADRG_Bitmap, BIT24);

/* If decompression was successful, optimize the image from 24-bit color   */
/* down to 8-bit color.  If successful, write the 8-bit palette to the     */
/* file 'Palette.mwc'.                                                      */
        if( ret_val == SUCCESS ){
           ret_val = L_OptimizeBitmap(&ADRG_Bitmap, &New256_Color,
                                 NO_DITHERING, 256);
           L_FreeBitmap(&ADRG_Bitmap);
           printf( " SUCCEEDED!\n" ) ;
        }
        else
           printf( " FAILED!\n" ) ;

/* Check for the proper number of colors written to the 'Palette.mwc' file.*/
        if( write(write_handle, New256_Color.pPalette, 768) != 768 ){
           printf( "Error writing 768 bytes!\n" ) ;
        }

/* Free the bitmap.                                                         */
        L_FreeBitmap(&New256_Color);
      }
    }

/* Close the file 'Palette.mwc'.                                            */
    close(write_handle);

/* Initialize and allocate memory for a bitmap.                             */
    L_InitBitmap(&Old_Palette, 256, 25, 24);
    ret_val = L_AllocateBitmap(&Old_Palette, TYPE_NOCONV);

/* Open the 'Palette.mwc' file.                                             */
    read_handle = open("Palette.mwc", O_RDONLY | O_BINARY, S_IREAD);

/* Place all 25 palettes (5x5 tiles) into the newly allocated bitmap       */
    for(cnt = 0; cnt < 25; cnt++){
     ret_val = read(read_handle, pRow, 768);
     ret_val = L_PutBitmapRow(&Old_Palette, pRow, cnt, 768);
    }

/* Close the 'Palette.mwc' file.                                            */
    close(read_handle);

/* Change the color byte order from RGB to BGR.                             */
    Old_Palette.Order = ORDER_BGR;

/* Optimize the palette file 'Palette.mwc' so as to produce the best 256   */
/* colors for all the images.                                               */
    ret_val = L_OptimizeBitmap(&Old_Palette, &New_Palette, NO_DITHERING,
          256);

   /* Free the Old_Palette bitmap.                                          */
        L_FreeBitmap(&Old_Palette);

   /* Set display resolution for the map image to 640x480.                  */
        L_SetVGASize( SIZE_640x480 );

   /* Extract row and column information from the center tile name.         */
        for(index = 5; index < 8; index++)
          row_num[index - 5] = center_tile[index];
        for(index = 9; index < 12; index++)
          col_num[index - 9] = center_tile[index];
        tile_row_cnt = atoi(row_num);
        tile_col_cnt = atoi(col_num);

   /* Load the display_tile array with the longitudinal and latitudinal     */
   /* values that make-up the center_tile name.                             */
        for(index = 0; index < 5; index++)
          display_tile[index] = center_tile[index];

   /* This loop is used to display a 5x5 matrix of tiles that will be       */
   /* centered around the tile input by the user.  The first tile processed */
   /* is in the upper left of the image.  The tiles along that row are then */
   /* processed sequentially and the next row is then processed starting from */
   /* the left side and working right.                                      */
        for(i = tile_row_cnt - 2; i <= tile_row_cnt + 2; i++){
          itoa(i, row_num, 10);
          if((i < 0) || (i > image_height)){
             for(index = 5; index < 8; index++)
                display_tile[index] = '?';
          }
          else if(i <= 9){
             display_tile[5] = '0';
             display_tile[6] = '0';
             display_tile[7] = row_num[0];
          }
          else if(i <= 99){
             display_tile[5] = '0';
             display_tile[6] = row_num[0];
             display_tile[7] = row_num[1];
          }
          else{
             for(cnt = 5; cnt < 8; cnt++)
                display_tile[cnt] = row_num[cnt - 5];
          }

   /* If the tile is out of bounds (ie. the column number is less than 0 and */
   /* greater than the image_width) a '?' is loaded into the file name so as */
   /* to produce a nonexistent tile name that will not be displayed.         */
          for(j = tile_col_cnt - 2; j <= tile_col_cnt + 2; j++){
             itoa(j, col_num, 10);
             display_tile[8] = '.';
             if((j < 0) || (j >= image_width)){
```

```
            for(index = 9; index < 12; index++)
              display_tile[index] = '?';
         }
         else if(j <= 9){
            display_tile[9] = '0';
            display_tile[10] = '0';
            display_tile[11] = col_num[0];
         }
         else if(j <= 99){
            display_tile[9] = '0';
            display_tile[10] = col_num[0];
            display_tile[11] = col_num[1];
         }
         else{
            for(cnt = 9; cnt < 12; cnt++)
              display_tile[cnt] = col_num[cnt - 9];
         }
         display_tile[12] = '\0';

         for(cnt = 0; cnt < 13; cnt++)
            dcom_tile[cnt] = display_tile[cnt];

/* Close cell_name array with the end-of-string character '\0'.          */
         display_tile[0] = first_char[0];

/* Initialize a bitmap.                                                  */
         L_InitBitmap(&ADRG_Bitmap, 128, 128, 24);

/* Decompress the map tile once again and store it in the bitmap.        */
         if((ret_val = L_DecompressBitmap( display_tile, &ADRG_Bitmap,
              BIT24)) <=0){
            printf("Decompression on file %s failed. %d\n", dcom_tile,
                 ret_val);
         }

/* Optimize the colors of the decompressed bitmap.                       */
         if( ret_val == SUCCESS ){
            ret_val = L_OptimizeBitmap(&ADRG_Bitmap, &New256_Color,
                 NO_DITHERING, 256);
            L_FreeBitmap(&ADRG_Bitmap);
         }

/* Load the optimized image palette (from 'Palette.mwc') into the newly  */
/* created optimized palette for the image.                              */
         for(cnt = 0; cnt < 768; cnt++)
            New256_Color.pPalette[cnt] = New_Palette.pPalette[cnt] ;

/* Re-index the 8-bit map data to the updated color palette.  This re-maps */
/* all pixel data (128x128 pixels) to the image palette.                 */
         L_GetBitmapRow(&New_Palette, new_pBuf, p_row_cnt, 256);
         p_row_cnt++;
         for(index = 0; index < 128; index++){
            L_GetBitmapRow(&New256_Color, pBuf, index, 128);
            for(cnt = 0; cnt < 128; cnt++){
             color_index = pBuf[cnt];
             pBuf[cnt] = new_pBuf[color_index];
           }
           L_PutBitmapRow(&New256_Color, pBuf, index, 128);
        }

/* Calculate the display coordinates for each tile and load the tile to    */
/* the screen.  Free the bitmap after each tile is displayed.              */
        if( ret_val == SUCCESS ){
           y_coord = y_center + ((tile_row_cnt - i) * 128);
           x_coord = x_center - ((tile_col_cnt - j) * 128);
           New256_Color.XOffset = x_coord ;
           New256_Color.YOffset = y_coord ;
           New256_Color.ViewPerspective = BOTTOM_LEFT ;
           L_ViewBitmapScreen( &New256_Color, 0, 0 ) ;
           L_FreeBitmap(&New256_Color);
        }



/* Free the optimized image color palette.                                 */
    L_FreeBitmap(&New_Palette);

/* Wait for a user input before clearing the map display.                  */
    getchar() ;

/* Reset monitor to original video mode.                                   */
    L_SetVideoMode(mode, page);
}
```

APPENDIX B

TECHNICAL SOURCES

[1] Digital Compression and Coding of Continuous-tone Still Images. ISO/IEC DIS 10918-1.

[2] C-Cube CL550 JPEG Image Compression Processor Data Book. C-Cube Microsystems, Inc., 1778 McCarthy Boulevard, Milpitas, CA 95035: February 1992.

MIL-A-89007: Military Specification ARC Digitized Raster Graphics. Defense Mapping Agency, 8613 Lee Highway, Fairfax, VA 22031-2137: February 1990.

LEADTOOLS Developers Toolkit. LEAD Technologies, Inc., 8701 Mallard Creek Road, Charlotte, NC 28262: 1992.

"Video Compression Technology Overview," Integrated Information Technology, Inc., 2445 Mission College Boulevard, Santa Clara, CA 95054: September 1991.

Purcell, Stephen C., "The C-Cube CL550 JPEG Image Compression Processor," C-Cube Microsystems, Inc.

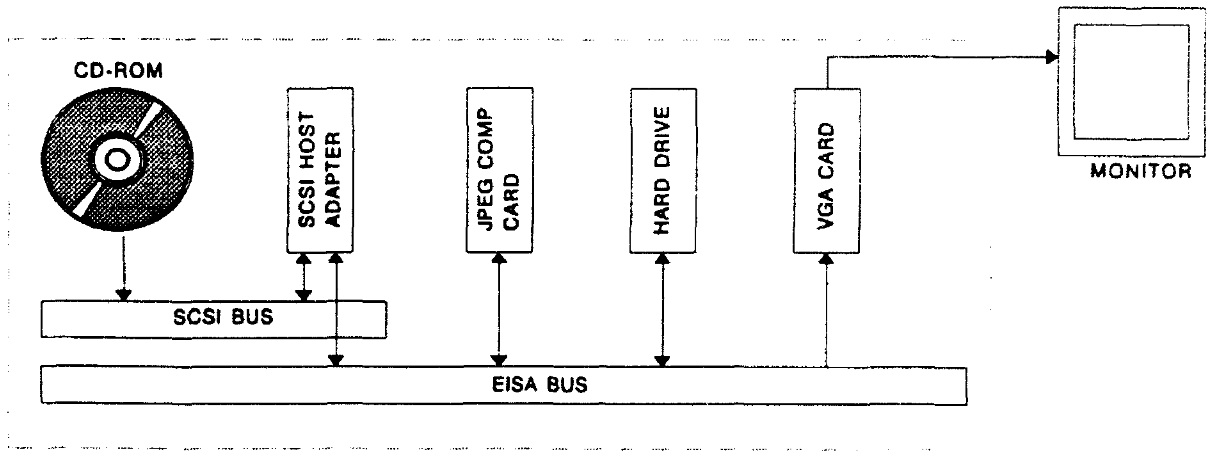
APPENDIX C

KEY TERMS

ADRG:	ARC Digitized Raster Graphics
BAUD:	Bits per Second
CCITT:	Consultative Committee on International Telephony and Telegraphy
CD-ROM:	Compact Disk Read-Only-Memory
CMYK:	Cyan, Magenta, Yellow, Black
DCT:	Discrete Cosine Transform
DIS:	Draft International Standard
DMA:	Defense Mapping Agency
EISA:	Extended Industry Standard Architecture
FOG-M:	Fiber-Optic Guided Missile
IDCT:	Inverse Discrete Cosine Transform
IEC:	International Electrotechnical Commission
IFCS:	Improved Fire Control System
I/O:	Input/Output
ISA:	Industry Standard Architecture
ISO:	International Standards Organization
JPEG:	Joint Photographic Experts Group
MLRS:	Multiple Launch Rocket System
MPEG:	Motion Picture Experts Group
MS-DOS:	Microsoft Disk Operating System
PC:	Personal Computer
Px64:	CCITT H.261 Standard for Video Teleconferencing
RGB:	Red, Green, Blue
SCSI:	Small Computer System Interface
TRIS:	Technical Risk Investigation System
UAV:	Unmanned Aerial Vehicle
UGV:	Unmanned Ground Vehicle
VGA:	Video Graphics Adapter
YUV:	Luminance, chrominance

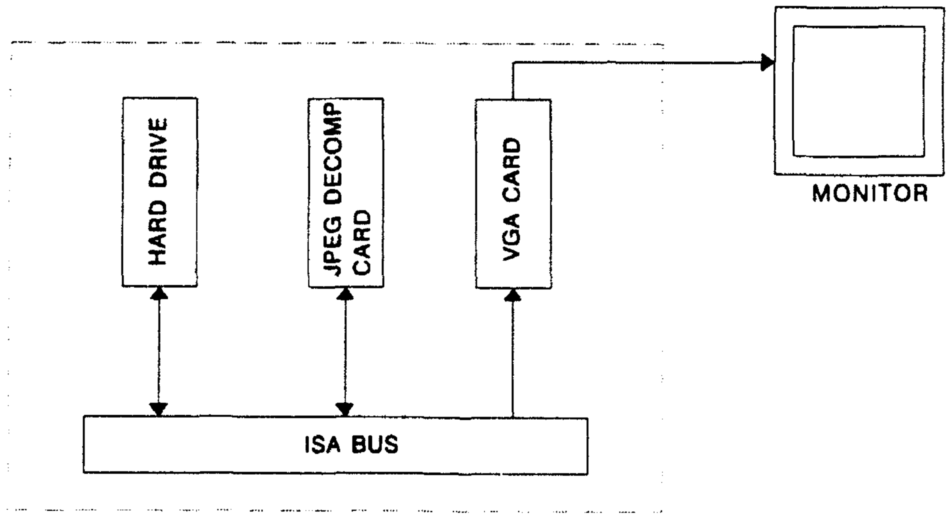
APPENDIX D

COMPRESSION HARDWARE DIAGRAM



486 PC

DECOMPRESSION HARDWARE DIAGRAM



486 PC

INITIAL DISTRIBUTION

	Copies
U.S. Army Materiel System Analysis Activity ATTN: AMXSY-MP Aberdeen Proving Ground, MD 21005	1
IIT Research Institute ATTN: GACIAC 10 W. 35th Street Chicago, IL 60616	1
AMSMI-RD	1
AMSMI-RD-AS-OG, Mr. W. E. Miller, Jr.	1
AMSMI-RD-AS-RA, Mr. Robert Eison	1
AMSMI-RD-AS-SS, Mr. Jonn Hatcher	1
AMSMI-RD-BA, Mr. Willie Fitzpatrick	1
AMSMI-RD-BA-C3I, Mr. Gary Clayton	1
AMSMI-RD-CS-R, Building 4484	5
AMSMI-RD-CS-T	1
AMSMI-RD-GC, Dr. Paul Jacobs	1
AMSMI-RD-GC-C, Mr. Carl Warren	1
AMSMI-RD-GC-L, Mr. David Jones	1
AMSMI-RD-GC-N, Mr. James McLean	1
AMSMI-RD-GC-S, Mr. Marc Crooks	2
Mr. Gerald Scheiman	1
Mr. Michael Pitruzzello	
Mr. Dan Reed	1
AMSMI-RD-GC-T, Mr. Ron Wicks	1
Mr. James Bradas	1
AMSMI-RD-SS, Dr. Kelly Grider	1
AMSMI-RD-SS-HW, Dr. K. L. Hall	1
AMSMI-RD-SS-SP-AM, Mr. Tom Smith	1
AMSMI-GC-IP, Mr. Bush	1
SFAE-CC-AD, COL Daniel Montgomery	1
Mr. Reginald Skinner	1
SFAE-CC-AD-TM, Mr. Fred Pera	1
SFAE-CC-AD-TM-SE, Mr. Don Fullerton	1
Ms. Susan Roberts	1
SFAE-FS-ML-TM, Mr. Billy Crosswhite	1
Mr. Frank Gregory	1
Mr. Ken Miller	1
Mr. Bob Wilkes	1